分散システムにおける資源割り当てアルゴリズム(計算量理論)

AUTHOR(S):

角川, 裕次; 山下, 雅史

CITATION:

角川, 裕次 ...[et al]. 分散システムにおける資源割り当てアルゴリズム(計算量理論). 数理解析研究所講究録 1994, 871: 168-174

ISSUE DATE:

1994-05

URL:

http://hdl.handle.net/2433/84040

数理解析研究所講究録
第871巻 1994年 168-174

分散システムにおける資源割り当てアルゴリズム

広島大学工学部第二類　角川裕次 (Hirotsugu Kakugawa)
広島大学工学部第二類　山下雅史 (Masafumi Yamashita)

Abstract

本稿では, 分散システムでの共有資源の割り当て問題を考察する. プロセス毎に使用可能な共有資源が異なる, というモデルに対する資源割り当て問題を解くために, 局所コータリーという構造を新たに提案する. また, 局所コータリーを用いた資源割り当てを行なう分散アルゴリズムを提案し, その正当性を示す.

1 はじめに

複数のプロセスが地理的に分散しそれらが互いに通信をしながら情報交換を行なう分散システムに於ける資源の共有を考える. 共有される資源は排他的な使用, 即ち2つ以上のプロセスは1つの資源を同時に使用する事が出来ないと仮定する. 資源が分散システム全域で共有される場合は従来からよく研究されてきている. 例えば, 分散データベースでは, データ項目を1つの共有資源と見做すことが出来る. そして, データの更新を行なう為に資源を確保すれば, 他のプロセスが資源を確保していない事が保証されるので, データの一貫性を失うことなく更新作業を行なう事が出来る.

従来の研究では, 1つの資源が分散システム全域で共有されるという資源共有のモデルが取り扱われ, 資源へのアクセスを行なう (臨界領域にある) プロセスの数を各時点において高々1つに制限する分散相互排除問題として研究されてきた [5][11][13][6]. Lamport [5], Ricart [11] によって提案された分散相互排除アルゴリズムは, 相互排除を行なう度に全てのプロセスにメッセージを送らねばならず, メッセージ数や耐故障性の点より好ましくない. Garcia-Molina ら [1] はコータリー (coterie) という構造を提案し, メッセージを送るべきプロセスの集合が必ずしも全てのプロセスではなく, 一部のプロセスのみでも相互排除を保証することができる事を示した. 前川 [6] は, 相互排除を行なうために送るべきメッセージ数が $O(\sqrt{n})$ で済むコータリーの構成法と分散相互排除アルゴリズムを示した. ここで n は分散システムに含まれるプロセスの数である.

本稿では, 資源が複数存在し, 資源を共有しているプロセスの集合が資源毎に異り, しかもプロセスが資源を必要とする時はその時点で使用可能な資源の内いずれでも割り当てられれば良い, という問題を考える. また, 資源には一意な名前が付けられており, 割り当てられた資源名をプロセスは知る事が出来るようにする, という条件も加える. [9], [15] では資源割り当て問題を議論し, アルゴリズムを提案しているが, いずれもシステム内で割り当てられた資源の数が資源総数を越えない事しか保証しておらず, 資源を確保したプロセスはどの資源を得たのかは知る事が出来ない.

第2節では, 本稿で取り扱う分散システムのモデル, 資源共有のモデル, 取り扱う問題などの定義を行なう. そして, 局所コータリーという新たな構造を提案する. 第3節では, 局所コータリーを用いた, 分散資源割り当てアルゴリズムを提案し, 第4節ではその正当性を証明する. 最後に第6節で本稿のまとめを行ない, 今後の課題について言及する.

2 準備

本稿では, 以下のことを仮定する. 分散システムは, n 個のプロセス $U=\{u_1,u_2,...,u_n\}$ と, これらで共有される資源の集合 $R=\{r_1,r_2,...,r_m\}$ より構成される.

プロセスとプロセス間の情報交換について, 以下の仮定おく. 各プロセスの実行速度は零以外であり, 各プロセスで共通な時計は存在しない. プロセス間の情報の交換はメッセージ通信のみで行なわれる. 各プロセスは, 任意のプロセスとメッセージの直接の交換が可能であり (完全ネットワーク), メッセージの追い越しは生じないとする. 各プロセスにはメッセージの受信待ち行列があり, メッセージは到着順に待ち行列に入れられる. プロセス及び通信路に関して, 故障は生じないと仮定する.

定義 1 関数 $\alpha: U \mapsto 2^R$ は, 各プロセスの使用可能な資源集合を表す写像である. 混乱のない限りこれを拡張し $V \subseteq U$ に対して $\alpha(V)$ と書く時は, $\alpha(V)=\cup_{v\in V}\alpha(v)$ を表すものとする. □

定義 2 $c=\langle R_1,R_2,...,R_n\rangle$, $R_i \subseteq R$ を**状況**といい, C を全ての状況の集合とする. プロセス u_i が資源の集合 R_i を使用している (割り当てられている) ときの状況を $\langle R_1,R_2,...,R_n\rangle$ と表す. □

定義 3 ρ を $U \times C$ から 2^R への写像で, 各 $c = \langle R_1, R_2, ..., R_n \rangle \in C$ $(R_i \subseteq R)$ に対し $\rho_{u_i}(c) = R_i$ と定める. (直観的には, $\rho_{u_i}(c)$ は, 状況 $c \in C$ においてプロセス $u_i \in U$ に割り当てられている資源の集合を表す.) □

定義 4 有限または無限の任意の状況の列 $\pi(c_0) = c_0, c_1, c_2, ...$ を c_0 より始まる遷移列と呼び, c_0 より始まる遷移列全ての集合を $\Pi(c_0)$ とおく. **資源競合回避問題** (*resource conflict resolution problem*) とは, 状況 $c_0 \in C$ と, システムの振舞いが以下の条件を満たす c_0 より始まる遷移列の集合 $\Lambda(c_0)$ に含まれるようにシステムの動作規定を定める問題である.

資源条件 全ての $\lambda = c_0, c_1, c_2, ... \in \Lambda$, i, $V \subseteq U$ に対し, 次の資源配分正当性が成立する. **資源配分正当性**: $\cup_{v \in V} \rho_v(c_i) \subseteq \alpha(V)$, かつ, 各 $u, v \in V$ $(u \neq v)$ に対し $\rho_u(c_i) \cap \rho_v(c_i) = \emptyset$

公平性条件 資源配分正当性を満たす c_0 より始まる, 任意の有限長の遷移列 $\pi = c_0, c_1, c_2, ..., c_k$ と, 全ての i, 全ての $V \subseteq U$ に対し,

$$\cup_{v \in V} \rho_v(c_l) = \alpha(V)$$

となる (π を接頭辞とする) 有限の長さの遷移列 $\lambda = c_0, c_1, c_2, ..., c_k, ..., c_l$ $(l \geq k)$ が存在する. □

ここで挙げた 2 つの項目は, 各々直観的いえば, 次のものである.

- どのプロセス u_i も, u_i が使用可能な資源 $= \alpha(u_i)$ のみしか使用しない.
- 任意の状況であっても, どのプロセスも資源を要求すれば, 有限時間内に資源を割り当てられる.

資源が 1 つのみで, かつ全てのプロセスがその資源を共有している場合については, 従来からよく研究されてきている. そのような共有のモデルのもとで, 資源割り当てを行なうために, **コータリー** (*coterie*) という構造が提案されている [1]. 集合 $U = \{u_1, ..., u_n\}$ の下でのコータリー $\mathcal{Q} = \{q_1, q_2, ...\} \subseteq 2^U$ とは, 以下の条件を満たすものをいう.

- Non-emptyness: $\forall q_i (q_i \in \mathcal{Q})[q_i \neq \emptyset]$
- Intersection Property: $\forall q_i, q_j (q_i, q_j \in \mathcal{Q})[q_i \cap q_j \neq \emptyset]$
- Minimality: $\forall q_i, q_j (q_i, q_j \in \mathcal{Q})[q_i \not\subseteq q_j]$

コータリーの導入の直観的な理由は以下の通りである. 資源を要求するプロセス u は, ある $q \in \mathcal{Q}$ を 1 つ選び, q に属する全てのプロセスに対して資源要求のメッセージを送る. 資源要求のメッセージを受けとったプロセスは, もし他のプロセスに資源使用の許可を送っていなければ, 資源使用許可を u に送る. u は, q の全てのプロセスから資源使用の許可を得ることができれば, 要求した資源を使うことができる. なお, 各 $q \in \mathcal{Q}$ のことを**コーラム** (*quorum*) と呼ぶ. この方法で資源が排他的に使用される理由は, 以下の通りである. Intersection Property により, 任意の 2 つのコーラムは共通のプロセスを持つ. また, 各プロセスは同時には 1 つのプロセスにしか資源の使用許可を与えない. 従って, この 2 点より, 2 つ以上のプロセスが同時にそれらが選んだコーラムの全てのプロセスから資源使用の許可が得られることはないからである.

コータリは, 資源が 1 つのみで, かつ全てのプロセスがその資源を共有している場合に対する, 資源割り当てを行なうための構造である. しかし, 本稿で取り扱う資源共有のモデルを考えると, コータリーでは不十分である. そこで, コータリーの概念を拡張した**局所コータリー** (*local coterie*) を提案する.

定義 5 集合 U と α に対して以下の条件を満たすときかつその時のみに限って $\mathcal{Q}$ は集合 U と α の下での**局所コータリー** (*local coterie*) である.

- 各 $u_i \in U$ に対し $\mathcal{Q}(u_i) \subseteq 2^U$
- Non-emptyness:
  $\forall i (1 \leq i \leq n)[\mathcal{Q}(u_i) \neq \emptyset]$
- Intersection property:
  各 $u_i, u_j \in U$ に対し, $\exists k (1 \leq k \leq m)[r_k \in \alpha(u_i) \cap \alpha(u_j)]$ ならば $\forall q_x \in \mathcal{Q}(u_i)$, $q_y \in \mathcal{Q}(u_j)[q_x \cap q_y \neq \emptyset]$ である.

- Minimality property:
  各 $u_i \in U$ に対し $\forall q_x, q_y \in \mathcal{Q}(u_i)[q_x \not\subseteq q_y]$ である.

局所コータリーの直観的説明は, 以下の通りである. u_i, u_j が少なくとも 1 つ資源を共有していれば, u_i, u_j のコーラムに交わりが存在しないといけない. 逆にいえば, 1 つも資源を共有していなければ, u_i, u_j のコーラムは互いに交わる必要はない. 資源を要求する時, u_i はあるコーラムの各プロセスに対して資源割り当て状況を問い合わせる. もし u_j と 1 つでも資源を共有していれば, コーラムの交わりに属するプロセスが資源割り当ての調停を行なえば良い. 一方, 資源を共有していなければ, 調停は必要なく, 独立に資源割り当てを行なって構わない.

$|R| = 1$ であり, かつ各 u_i に対し $\alpha(i) = R$ であれば, 局所コータリーは従来のコータリーと同一であることに注意せよ. この意味で局所コータリーはコータリーの拡張となっている.

3 資源割り当てアルゴリズム

資源割り当てを行なう分散アルゴリズムを提案する. アルゴリズムを示す前に, アルゴリズムの概要を説明する.

3.1 アルゴリズムの概要

U と α の下での局所コータリーを $\mathcal{Q}$ とする. 各プロセス $u_i \in U$ は, 分散データベースの断片を保持しておく. u_i が保持すべき情報は, ある q に対して $u_i \in q(\in \mathcal{Q}(u_j))$ であるプロセス u_j の共有している資源である. すなわち u_i は, u_i をコーラムのメンバーとして持つプロセス u_j の共有する資源について (分散) 管理を行なう. 複数のプロセスが 1 つの資源を管理することに注意せよ. u_i の管理する資源の集合を $S_i \subseteq R$ とする. u_i は各 $r \in S_i$ に対し, r が割り当てられているプロセス名 (未割り当てならば $\perp$) と, 割り当てられた時刻 (未割り当てならば最後にそれが解放された時刻) をデータベースの項目として保持しておく.

プロセス u_i において k個 $(1 \leq k \leq |\alpha(u_i)|)$ の資源要求が生じると, あるコーラム $q \in \mathcal{Q}(u_i)$ を 1 つ選び, q に属する全てのプロセスにデータベースへの問い合わせメッセージ ⟨QUERY⟩ を送る. ⟨QUERY⟩ を受けたプロセス u_j は, u_j のデータベース断片を参照し, $\alpha(u_i)$ に属する各資源の項目値をメッセージ RESPONSE に入れて u_i に送る. u_i は q に属する全てのプロセスより RESPONSE が返るのを待つ. u_i は各 RESPONSE で得られたデータベース断片より, k個の未割り当ての資源 $s_1, s_2, ..., s_k \in R$ を見つる. そして, それらの資源を使用することを q の全てのプロセスにメッセージ ⟨LOCK⟩ により通知する. 通知を受けた qの各プロセスは, $s_1, s_2, ..., s_k$ が u_i に割り当て済みと分散データベース断片を更新する. データベースの一貫性保持のため, q の各プロセスはデータベース項目を u_i に送ってから資源の使用の通知を u_i より受けるまでは, 他の資源要求には応えない. 即ち, データベースへの問い合わせは排他的に行なわれる. プロセス u_i がデータベースへの問い合わせを行なう事の出来る時, u_i はデータベースへのアクセス権を持つ, という. u_i が資源 $s_1, s_2, ..., s_k$ の使用を終了したら, q の各プロセスにメッセージ ⟨UNLOCK⟩ を送ることで資源の解放を知らせる. 資源の解放を知った q の各プロセスは, $s_1, s_2, ..., s_k$ が未割り当てであるとデータベース項目を更新する.

以上で述べた方法では, デッドロックや飢餓状態が生じてしまう. Lamprot [5] による論理時間に基づいた時刻印を用いて資源要求に優先順序を付けることで, これらを避けることができる. また, 分散環境で大局的に一貫性のある時計が存在しないので, データベース中での資源の割り当てられた, あるいは資源が解放された時刻は, 論理時間を用いる. 以降, 特に断らない限り, 論理時間のことを時間と呼ぶ. アルゴリズムを以下に示す.

3.2 アルゴリズム

資源要求に優先度をつけるために, Lamport [5] による時刻印を用いる. 各プロセス $u_i \in U$ は, 以下の局所変数を持つ.

- $t(u_i)$ — u_i が資源の要求を行なった時の時刻印. 初期値は 0.
- S_i — u_i をコーラムの要素として持つプロセスの集合. $S_i = \{u_j \mid$ ある q に対し, $u_i \in q(\in \mathcal{Q}(u_j))\}$ と定められる.

- $D(i)$ — u_i の管理する分散データベース断片. $D(u_i)$ は, u_i が共有している各資源の割り当て状況の表であり, $\{\langle r_a, v_a, t_a\rangle, \langle r_b, v_b, t_b\rangle, ...\}$ の形をしている. 但し, $\{r_a, r_b, ...\} = \cup_{u_j \in S_i} \alpha(u_j)$, $v_a, v_b, ... \in S_i \cup \{\perp\}$, $t_a, t_b, ... \in \mathrm{N}$ である. なお, r_j がプロセス u_l に割り当てられていれば $v_j = u_l$ であり, どのプロセスにも割り当てられていなければ, $v_j = \perp$ である. t_j は, r_j が v_j に割り当てられた時の, u_iでの時刻である. もし r_j が未割り当てであれば, t_j は解放を行なったプロセスにおける, 解放を行なった時の時刻とする. 初期値は, 各 $r_j \cup_{u_j \in S_i} \alpha(u_j)$ に対し $\langle r_j, \perp, 0\rangle$ である.
- $W_L(u_i)$ — 資源要求に対して返事 RESPONSE を u_j 送ったが, まだそれに対応した $\langle$LOCK$\rangle$ を u_j から受けとっていなければ $W_L(u_i) = u_j$ である. そうでなければ, $W_L(u_i) = \perp$ とする.
- $W_Q(u_i)$ — 時刻印を優先度とする, 優先度付き待ち行列.

では, アルゴリズムを以下に示す.

- **プロセス u_i が k個の資源を要求する時.**
  $t(u_i)$ の値を現在の時計の値にする. あるコーラム $q \in Q(u_i)$ を選び, 各 $v_j \in q$ に対し $\langle$QUERY$, t(u_i)\rangle$ を送る. u_i は, 各 $v_j \in q$ が少なくとも 1 度は RESPONSE によってデータベース断片を送り, かつ, 互いに異なる k個の資源 $s_1, s_2, ..., s_k \in \alpha(u_i)$ が存在するまで待つ. (これは, 送られて来たデータベース項目より調べる.) そのような資源の集合が存在すると, t を現在の時刻とすれば, u_i は 各 $v_j \in q$ に対し $\langle$LOCK$, t, s_1, s_2, ..., s_k\rangle$ を送り, 資源の使用を知らせる.

  なお, データベース項目から, 資源 r が未割り当てであるか否かは, 以下のようにして判断する. 各プロセス $v_1, v_2, ... \in q$ の返してきた r に関する項目の時刻欄の値をそれぞれ $t_1, t_2, ...$ とし, $t = t_j = \max\{t_1, t_2, ...\}$ と定める. v_j の返してきた r の割り当て先プロセス名が $\perp$ であるときのみに限って, r は未割り当てであるとする.
- **プロセス u_i が資源を解放する時.**
  t を現在の時刻とし, 資源要求時に選んだコーラム q の各プロセスに $\langle$UNLOCK$, t, s_1, s_2, ..., s_k\rangle$ を送る.
- **プロセス u_i が u_j から RESPONSE を受けた時.**
  u_j からのデータベース項目を記憶する. (現在の資源要求に関して) すでに u_j からデータベース項目を受けとっていれば, 古い方は破棄する.
- **プロセス u_j が u_i より $\langle$QUERY$, t_i\rangle$ を受けた時**
  - $W_L(u_j) = \perp$ の時:
    u_i に $r \in \alpha(u_j)$ のデータベース項目を送り, $W_L(u_j)$ を u_i にする.
  - $W_L(u_j)$ が $u_l \in U$ の時:
    t_l を (u_j のデータベース断片中の) u_l の要求の時刻とする. u_j は u_i の時刻印 $\langle t_i, u_i\rangle$ と $\langle t_l, u_l\rangle$ を比較し, もし u_l が高い優先度を持つならば u_i の要求を待ち行列 $W_Q(u_j)$ に入れる. もしそうでないならば, u_l に送ったデータベース問い合わせの応答を取り消すために, u_l へ PREEMPT メッセージを送る. この後, u_l から $\langle$RETURN$, s_1, ...\rangle$ が返されるのを待つ. そして, 分散データベース断片の $s_1, ...$ の項目を未割り当てと更新し, RESPONSE に $r \in \alpha(u_j)$ のデータベース項目を入れて u_i に送り, $W_L(u_j)$ を u_i とする.
- **プロセス u_j が u_i より $\langle$LOCK$, t_j, s_1, ...\rangle$ を受けた時**
  $W_L(u_j)$ を $\perp$ とし, 分散データベース断片の各 $s_1, ...$ の項目を u_i に割り当て済みと更新する.
- **プロセス u_j が u_i より $\langle$UNLOCK$, t_i, s_1, ...\rangle$ を受けた時**
  まず, 分散データベース断片の各 $s_1, ...$ の項目を, 未割り当てと更新する.
  - ある $u_l \in U$ に対して $W_L(u_j) = u_l$ の時.
    この状況では u_l は必要なだけの資源が得られていない. $r \in \alpha(u_l)$ のデータベース項目を RESPONSE に入れて u_l に送る. (u_j が u_l に RESPONSE を送るのは 1 度とは限らないことに注意せよ.)
  - $W_L(u_i) = \perp$ のとき.
    もし待ち行列 $W_Q(u_j)$ が空でなければ, $W_Q(u_j)$ の先頭のプロセス u_l を取り出し, $W_L(u_j)$ の値を u_l とした後, $r \in \alpha(u_j)$ のデータベース項目を RESPONSE メッセージに入れて u_l に送る.

- **プロセス u_j が u_i より PREEMPT を受けた時**
  もしすでに資源を使用中であれば, 無視する. (資源解放時に ⟨UNLOCK⟩ が送られる.) そうでなければ, u_i に ⟨RETURN⟩ メッセージを送り, u_i から送られてきたデータベース項目を消去し, u_i からデータベース項目がまだ送られて来ていないとする.

4 正当性の証明

ここでは, 上で示したアルゴリズムが正しいことを証明する. なお, 資源を獲得したプロセスは有限時間内で解放を行なうと仮定する.

初期状況 c_0 は, どのプロセスも資源を獲得していないものとする. 資源要求ステップにおいて, あるコーラムの全てのプロセスから RESPONSE を受け取ってから ⟨LOCK⟩ を送るまでの領域を **Q-領域** といい, プロセス u が Q-領域の命令を実行しているならば, u は Q-領域にある, という.

次の補題は, データベースへの問い合わせが, 資源を共有しているプロセス間で排他的に行なわれることを示している.

補題 1 u_i, u_j を, ある資源 $r \in R$ が存在して $r \in \alpha(u_i) \cap \alpha(u_j)$ である任意の 2 つのプロセスとすると, 任意の時点において u_i, u_j は同時に Q-領域にない.

(証明) ある状況 c において, u_i, u_j がともに Q-領域にあると仮定すると, u_i (u_j) は $q_i \in \mathcal{Q}(u_i)$ $(q_j \in \mathcal{Q}(u_j))$ の全てのプロセスより RESPONSE を受け取っている. 各プロセスは, あるプロセスに RESPONSE を送ってから ⟨LOCK⟩ あるいは ⟨RETURN⟩ が返されるまでは他のプロセスへ RESPONSE を送らないので, $q_i \cap q_j = \emptyset$ である. しかし, u_i, u_j は共にある資源 $r \in R$ を共有しているので, 局所コータリーの定義より,

$$\forall q_i \in \mathcal{Q}(u_i),\ q_j \in \mathcal{Q}(u_j)[q_i \cap q_j \neq \emptyset]$$

であり, 矛盾. □

補題 2 任意の 2 つのプロセス u_i, u_j と任意の状況 $c \in C$ に対し, ある $r \in \alpha(u_i) \cap \alpha(u_j)$ が存在し, u_i が r を使用中, かつ, プロセス u_j が Q-領域にあると仮定し, t_i を u_i の資源要求に対する時刻印とする. $q_i \in \mathcal{Q}(u_i)$ $(q_j \in \mathcal{Q}(u_j))$ を, u_i (u_j) が資源要求を行なった際に選択したコーラムとする. このとき, 全ての $x \in q_i \cap q_j$ に対し, x の分散データベース断片の r に関する項目の値が $\langle r, u_i, t_i \rangle$ である.

(証明) あるプロセス $x \in q_i \cap q_j$ が存在し, x の分散データベース断片の r に関する項目の値が $\langle r, u_i, t_i \rangle$ 以外と仮定する. u_i は資源 r を使用中であるので, 使用に先だって q_i の全てのプロセスに ⟨LOCK⟩ メッセージを送っている. アルゴリズムの定義より, どのプロセスも, データベース項目を送った後は, ⟨LOCK⟩ または ⟨RETURN⟩ が返されるまでは他のプロセスには RESPONSE メッセージを送らない. よって, u_j にデータベース項目を送ったプロセス x は, それに先だって u_i より ⟨LOCK⟩ メッセージを受信をしている. よって, この時に x の分散データベース断片の r の項目値は $\langle r, u_i, t_i \rangle$ になる. これ以降, u_j が Q-領域に進むまでの間, r は u_i に割り当てられているので, その項目値は変わらない. よって矛盾. □

以下に, データベースの一貫性について定義しておく.

定義 6 分散データベースが局所コータリー $\mathcal{Q}$ の下で資源 $r \in R$ に関して**一貫性がある**とは, 以下が成立するときをいう.

任意の状況 $c \in C$, 任意の資源 $r \in R$, そして任意のプロセス $u \in U$ に対し, $r \in \rho_u(c)$, 即ち r が u に割り当てられているならば, ある $q \in \mathcal{Q}(u)$ が存在して全ての $v \in q$ の r に関する項目値が $\langle r, u, t \rangle$ である. (但し, t は u が資源を獲得した時の u での時刻である.)

全ての資源 $r \in R$ に対し, 分散データベースが局所コータリー $\mathcal{Q}$ の下で資源 r に関して一貫性があるとき, 分散データベースが局所コータリー $\mathcal{Q}$ の下で一貫性があるという. なお, 局所コータリー $\mathcal{Q}$ が文脈から明らかな時は, $\mathcal{Q}$ を省略して書く. □

補題 3 c_0 より始まる任意の遷移列 $c_0, c_1, c_2, \ldots \in \Pi(c_0)$ と全ての $i \geq 0$ に対し, 資源配分正当性 $\cup_{v \in V} \rho_v(c_i) \subseteq \alpha(V)$ が成立する.

(証明) 初期状況 s_0 では, どのプロセスも資源を使用していないので, 資源配分正当性が成立する. 補題 1より, Q-領域への進入は資源を共有しているプロセス間では排他的に行なわれる. この事と補題 2, ならびに局所コータリーの intersection property より, 分散データベースの一貫性がいえる.

どのプロセス u_i も, 未割り当てである資源のうち $\alpha(u_i)$ に属しているもののみを使用するので, ひとつの資源が複数のプロセスに割り当てられること, ならびに $\alpha(u_i)$ に属していない資源を使用することはアルゴリズムの定義より生じない. 従って, 本補題が成立する. □

定理 1 デッドロックは生じない.
(証明) 仮定より, どのプロセスも資源を獲得した後にさらに資源要求を行なわないので, ひとたびプロセスが資源を獲得した後はそれを解放するのみである. 従って, 更なる資源要求によるデッドロックは生じない. 故に, データベースへの問い合わせ段階でのデッドロックのみを考えればよい.

$\pi(c_0) = c_0, c_1, c_2, ...$ を, デッドロックが生じる遷移列と仮定する. プロセスがひとたびデッドロックになると, 以降は常にデッドロックであり続ける. プロセス数は有限なので, ある状況 c_d が存在し c_d 以降はデッドロック状態であるプロセスは一定となる. 以下では, c_d 以降を考える. なお, c_d 以降, メッセージのやりとりを全く行なわないプロセスが存在するかも知れないが, 一般性を失うことなくそのようなプロセスの存在は無視して考える.

全てのデッドロック状態のプロセス集合を $V = \{v_1, v_2, ..., v_l\} \subseteq U$, $l \geq 2$ とし, この中で資源要求に対する時刻印が最小である (すなわち, 最高の優先度を持つ) プロセスを, 一般性を失うことなく v_1 とする. v_1 のデータベース問い合わせメッセージ ⟨QUERY⟩ は, 有限時間内にあるコーラム $q \in \mathcal{Q}(v_i)$ の全てのプロセスにたどり着く. 時刻印は単調増加であるので, 有限時間内に v_1 の時刻印が最大の優先度を持つようになる. アルゴリズムの定義より, q の各プロセス $w \in q$ は, 以下の動作を行なう. もし, 他のプロセス w' に RESPONSE を送ったが, 対応した ⟨LOCK⟩ が返ってきていない場合は, w' に PREEMPT メッセージを送って, データベースへの問い合わせ権利を横取りし, v_1 にデータベースへの問い合わせ権利を譲る. もし他のプロセスが資源解放のメッセージ ⟨UNLOCK⟩ を w に返したなら, v_1 の優先度が一番高いので, w は v_1 にデータベース項目を送る. 以上のことが繰り返されるが, v_1 と資源を共有しているプロセスはこの間資源の獲得はできず, 資源を既に保持しているプロセスが資源の解放を行なうだけであるので, v_1 の要求している資源数の未割り当て資源が有限時間内に存在するようになり, しかもデータベース項目が v_1 に送られる. 従って, v_1 は要求した資源を獲得できる. これは矛盾. □

定理 2 資源を要求したどのプロセスも, 有限時間内に資源を獲得できる.
(証明) 定理 1と同様にして証明できる. 概要は以下の通りである. もし有限時間内に資源を獲得できないプロセス $u \in U$ が存在したと仮定する. 有限時間内に u の要求の時刻印は他のプロセスのものより優先度が高くなり, 資源割り当てが u に対して行なわれるようになり, 仮定に矛盾する. □

定理 3 そのアルゴリズムは, 資源競合回避問題を解く.
(証明) 補題 3 より, 初期状況 c_0 より始まる任意の遷移列 $\lambda = c_0, c_1, c_2, ...$ に対して $\cup_{v \in V} \rho_v(c_i) \subseteq \alpha(V)$ が成立する. また, 任意のプロセス u_i に対し $Q = \mathcal{Q}(u_i)$ とすると, 局所コータリーの構成により, 全てのプロセス $u_j \in \cup_{j \in Q} q$ は, $\alpha(u_i)$ に属する全ての資源の分散データベースの項目を管理している. 故に, $\alpha(u_i)$ 全ての資源が未割り当てであり, かつ他のどのプロセスも資源を要求していない状況で u_i が資源要求を行なえば, 定理 1, 定理 2 より, 任意の $R' \in \alpha(u_i)$ を有限時間内に獲得できる. よって, $V \subseteq U$ を任意のプロセス集合とすると, 資源状況の成立している任意の状況 c_i より, 全てのプロセスが資源を解放すれば状況は c_0 に遷移でき, ここより $\cup_{v \in V} \rho_v(c_l) = \alpha(V)$ なる状況 c_l に遷移できる. □

5 おわりに

本稿では, 資源毎に共有しているプロセス集合の異なるモデルでの, 資源割り当て問題について議論した. この問題を解くために, コータリーを拡張した局所コータリーを提案した. k-コータリーには, k個の互いに交わらないコーラムがあり, [3] で提案された k-コータリーに基づく分散 k-相互排除アルゴリズムでは, 臨界領域に進むためには空いたコーラムを順々に探して行く必要があった. 一方, 本稿で提案したアルゴリズムは, 最初にコーラムを 1 つ選んだ後は, それに対してのみメッセージをやりとりするだけで良い. この意味でも, 本稿のアルゴリズムは改良されている. 紙面の都合でメッセージ複雑度のの解析は示さなかったが, 本稿で示したアルゴリズムのメッセージ複雑度は, 最良時には $4|q|$, 最悪時には $(|\alpha(u_i)| + 7)|q|$ である.

本稿では局所コータリーを提案したが, 具体的な構成法については今後の課題とする. トリビアルな解決法としては, Garcia-Molina ら [1] によるコータリーをそのまま用いることが出来るが, それでは不必要に大きなコーラムとなってしまい, 局所コータリーの特質を生かしていない.

また, 茨木ら [2] は, コータリーをブール関数の観点よりその性質を調べており, 山下ら [14] は, k-コータリーをグラフ理論の観点よりその性質を考察している. 局所コータリーに対しても, 同様な観点より調べることは大変興味あることと思われる.

参考文献

[1] Hector Garcia-Molina and Daniel Barbara. How to assign votes in a distributed system. *Journal of the ACM*, Vol. 32, No. 4, pp. 841–860, October 1985.

[2] Toshihide Ibaraki and Tiko Kameda. Theory of coteries. In *Procs. 3rd Symp. on Paralleland Distributed Systems*, pp. 150–157, 1991.

[3] Hirotsugu Kakugawa, Satoshi Fujita, Masafumi Yamashita, and Tadashi Ae. A distributed k-mutual exclusion algorithm using k-coterie. *Information Processing Letters.* to appear.

[4] Hirotsugu Kakugawa, Satoshi Fujita, Masafumi Yamashita, and Tadashi Ae. Availability of k-coterie. *IEEE Transactions on Computers*, Vol. 42, No. 5, pp. 553–558, May 1993.

[5] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, Vol. 21, No. 7, pp. 558–565, July 1978.

[6] Mamoru Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, Vol. 3, No. 2, pp. 145–159, March 1985.

[7] Yoshifumi Manabe and Shigemi Aoyagi. A distributed k-mutual exclusion algorithm using k-coterie. *IEICE Japan, SIG Computation Record*, Vol. COMP91-13, pp. 11–18, May 1993. (in Japanese).

[8] Kerry Raymond. A distributed algorithm for multiple entries to a critical section. *Information Processing Letters*, Vol. 30, pp. 189–193, February 1989.

[9] Michel Raynal. A distributed solution to the k-out of-m resources allocation problem. In *Lecture Notes in Computer Science 497*, pp. 599–609. Springer-Verlag, 1991.

[10] Michel Raynal. A simple taxonomy for distributed mutual exclusion algorithms. *ACM Operating Systems Review*, Vol. 25, No. 2, pp. 47–51, 1991.

[11] Glenn Ricart and Ashok K. Agrawala. An optimal algorithm for mutual exclusion in computer network. *Communications of the ACM*, Vol. 24, No. 1, pp. 9–17, January 1981.

[12] Pradip K. Srimani and Rachamallu L.N. Reddy. Another distributed algorithm for multiple entries to a critical section. *Information Processing Letters*, Vol. 41, No. 1, pp. 51–57, January 1992.

[13] Ichiro Suzuki and Tadao Kasami. A distributed mutual exclusion algorithm. *ACM Transactions on Computer Systems*, Vol. 3, No. 4, pp. 344–349, November 1985.

[14] Masafumi Yamashita and Tiko Kameda. Greedy graphs for independence number: A graph theoretical study on k-coteries. memo, August 1993.

[15] 真鍋義文, 青柳滋己. 分散 k-相互排除問題について. 電子情報通信学会コンピュテーション研究会, Vol. COMP 91-13, pp. 11–18, May 1993.